Intuiting Predictive Algorithms

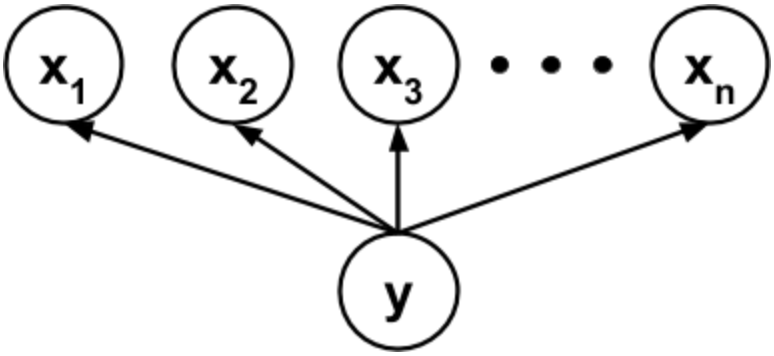
Justin Skycak, 2018

The goal of this write-up is to show how various predictive algorithms function and relate to each other.

1. Naive Bayes

If we know the causal structure between variables in our data, we can build a Bayesian network, which encodes conditional dependencies between variables via a directed acyclic graph. Such a model is constrained by our human understanding of the relationship between parts of the data, though, and may not be optimal when we wish to predict a target variable despite knowing little about the other variables to which it may or may not relate.

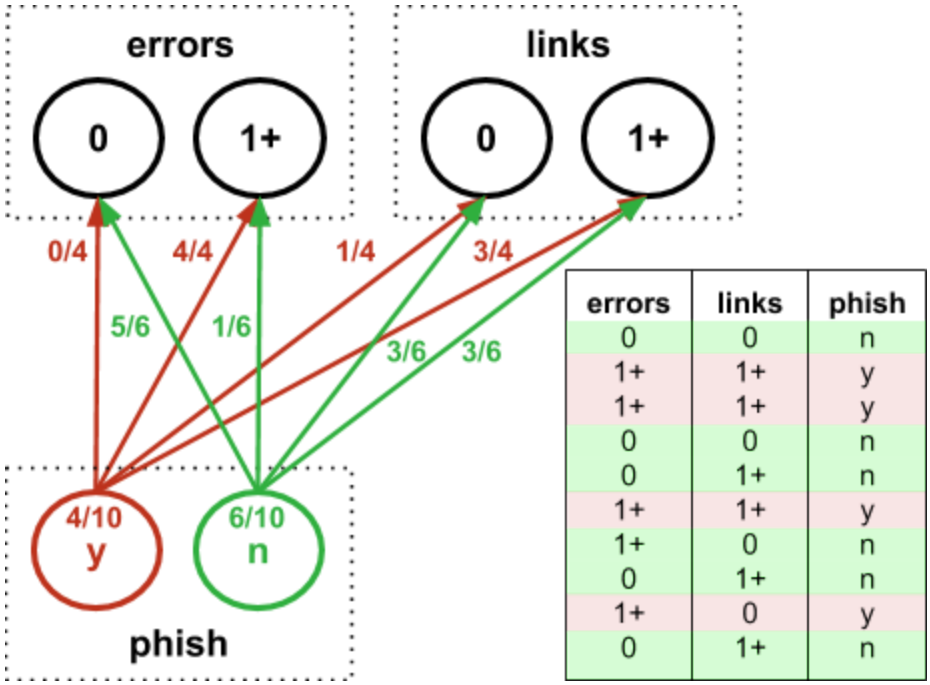
That being said, if we know that the target variable is a class that somehow encapsulates the other variables, it can be worthwhile to try a Bayesian network where the other variables are assumed to depend conditionally and independently on the class. This is called Naive Bayes classification because it naively assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.



If the data is given by $\{((x_{ij}), y_i)\}$ and each y_i belongs to a class C_k, then the Naive Bayes classifier computes

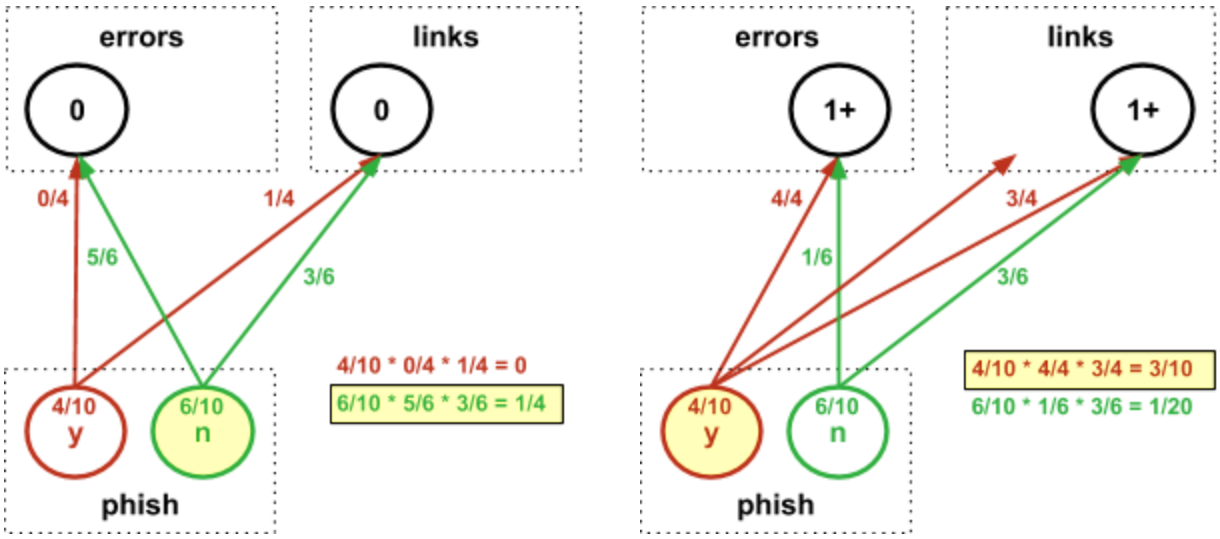
$$\begin{aligned} C_k &= \arg\max_{C_k} P(C_k|x) \\ &= \arg\max_{C_k} \frac{P(x|C_k)P(C_k)}{P(x)} \\ &= \arg\max_{C_k} P(x|C_k)P(C_k) \\ &= \arg\max_{C_k} P(C_k) \prod_j P(x_j|C_k) \end{aligned}$$

For example, we could build a Naive Bayes classifier to predict whether an email is a phishing attempt based on whether it has spelling errors and links:



errors	links	phish
0	0	n
1+	1+	y
1+	1+	y
0	0	n
0	1+	n
1+	1+	y
1+	0	n
0	1+	n
1+	0	y
0	1+	n

We could then use our model to test whether a new email is a phishing attempt:



In this example, we used discrete bins for the features -- but Naive Bayes can also handle features which are fit to continuous distributions. And despite assuming that features are independent (and thus potentially ignoring a lot of useful information), Naive Bayes can sometimes perform well enough in simple applications to get the job done.

2. MAP and MLE

Given data $D = \{((x_{ij}), y_i)\}$, if we model the relationship between the predictors and the target as being governed by parameters $\theta = (\theta_i)$, then Bayes' rule tells us that

$$p((x, y)|D) = \int p((x, y)|\theta)p(\theta|D)d\theta$$

We can interpret the integral as an average over all models, where the weight of a model's contribution to the sum is governed by the $p(\theta|D)$ term. This term is called the posterior or "a posteriori" distribution, as it is the result of updating the prior or "a priori" distribution $p(\theta)$ (which reflects our previous beliefs about the parameters) with the information that the data tells us.

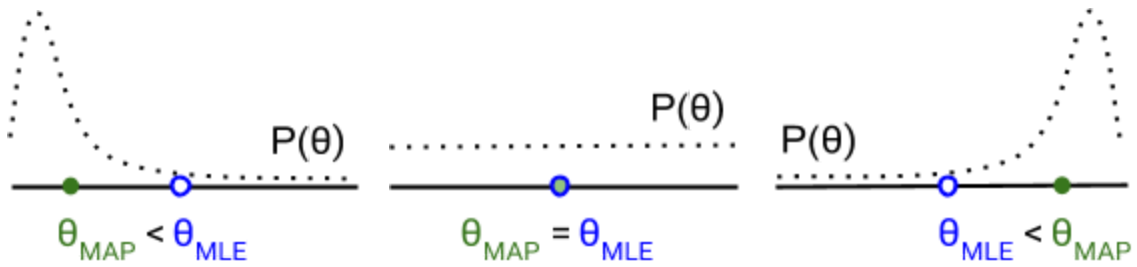
The average is difficult to compute, since the number of models grows exponentially with the number of parameters. It is easier to just pick the model with the maximum a priori distribution, rather than averaging over the entire ensemble. This is called Maximum A Posteriori (MAP) estimation.

$$
\begin{aligned}
\theta_{MAP} &= \arg\max_{\theta} p(\theta|D) \\
&= \arg\max_{\theta} \frac{p(D|\theta)p(\theta)}{p(D)} \\
&= \arg\max_{\theta} p(D|\theta)p(\theta) \\
&= \arg\max_{\theta} p(\theta) \prod_{i} p(y_i|x_i, \theta) \\
&= \arg\max_{\theta} \left(\log p(\theta) + \sum_{i} \log p(y_i|x_i, \theta) \right)
\end{aligned}
$$

If we want to model as though we know nothing aside from what the data tells us, then we can use the Jeffreys prior, which assigns $p(\theta)$ as a uniform distribution and is also known as the "uninformative" or "improper" prior since it does not actually depend on θ. When we perform MAP estimation using the Jeffreys prior, we are doing what is known as Maximum Likelihood Estimation (MLE). MLE derives its name from the fact that MAP with the Jeffreys prior amounts to maximizing $p(D|\theta)$, which is known as the likelihood.

$$
\begin{aligned}
\theta_{MLE} &= \arg\max_{\theta} p(\theta|D) \\
&= \arg\max_{\theta} \frac{p(D|\theta)p(\theta)}{p(D)} \\
&= \arg\max_{\theta} p(D|\theta) \\
&= \arg\max_{\theta} \prod_{i} p(y_i|x_i, \theta) \\
&= \arg\max_{\theta} \sum_{i} \log p(y_i|x_i, \theta)
\end{aligned}
$$

To visualize the relationship between the MAP and MLE estimations, one can imagine starting at the MLE estimation, and then obtaining the MAP estimation by drifting a bit towards higher density in the prior distribution.

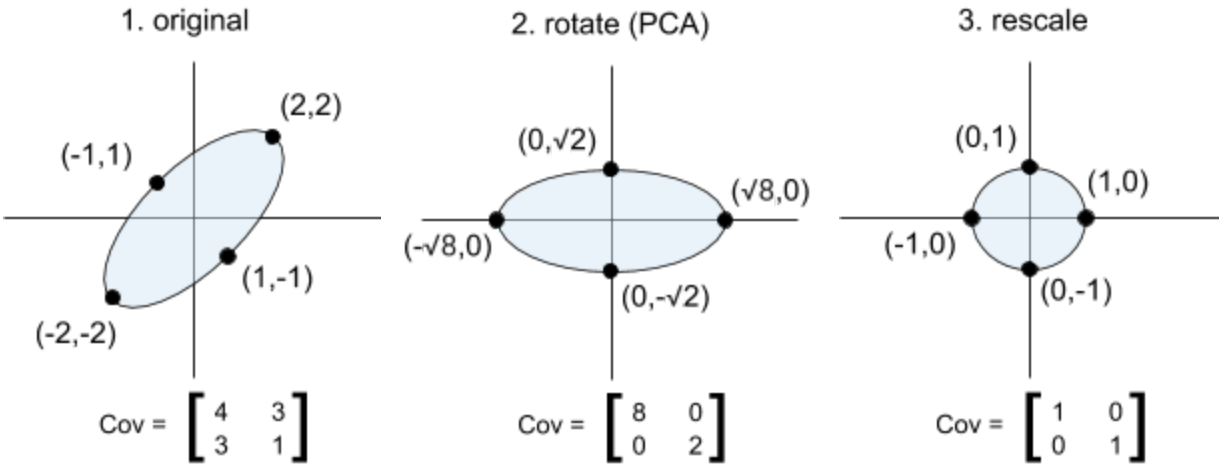


3. Linear Regression

In linear regression, we model the target y as a random variable whose expected value is depends on a linear combination of the predictors x (including a bias term, i.e. a column of 1s). When the noise is assumed to be Gaussian, MLE simplifies to least-squares:

$$\begin{aligned}\theta_{ML} &= \arg\max_\theta \sum_i \log N(y_i|\mu = \theta^T x_i, \sigma^2) \\ &= \arg\max_\theta \sum_i \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}\left(y_i - \theta^T x_i\right)^2\right) \\ &= \arg\min_\theta \sum_i \left(y_i - \theta^T x_i\right)^2\end{aligned}$$

In multivariate linear regression, each y_i is a vector containing multiple targets y_{ij}. If the covariance matrix of the targets is a multiple of the identity matrix, then Gaussian MLE again simplifies to least squares. Provided the targets are linearly related, we can cause the covariance matrix to become a multiple of the identity matrix by converting the targets to an orthonormal basis of principal components (this is known as PCA, or principal component analysis).



One benefit of linear regression over more complex models is that linear regression is very interpretable. Provided the predictors are normalized and are not linearly dependent, the parameter or coefficient for a particular term can be interpreted as its “weight” in determining the prediction. Even if the predictors are linearly dependent, we can still make the model interpretable if we replace the predictors with a subset of their principal components before performing the regression. This is called Principal Component Regression.

Some other types of linear regression include polynomial, logistic, and regularized (ridge) regression. In polynomial regression, we include not just x_i, but also x_i^2, x_i^3, etc. as predictors. In logistic regression, where the target is binary, we model the target as a Bernoulli random variable where the log of the odds ratio of the success probability is given by a linear regression:

$$\begin{aligned} p(y|x,\theta) &= B\left(x \middle| \log\left(\frac{p}{1-p}\right) = \theta^T x\right) \\ &= B\left(x \middle| p = \frac{1}{1+\exp(-\theta^T x)}\right) \end{aligned}$$

In regularized or “ridge” regression, we assume a prior other than the Jeffreys prior. A Gaussian prior gives rise to L2 regularization:

$$\begin{aligned} \theta_{MAP} &= \arg\max_\theta \left(\log N(\theta|\mu=0,\sigma^2) + \sum_i \log p(y_i|x_i,\theta)\right) \\ &= \arg\max_\theta \left(\log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}||\theta||^2\right) + \sum_i \log p(y_i|x_i,\theta)\right) \\ &= \arg\max_\theta \left(-\frac{1}{2\lambda^2}||\theta||^2 + \sum_i \log p(y_i|x_i,\theta)\right) \end{aligned}$$

A Laplacian prior gives rise to L1 regularization:

$$\begin{aligned} \theta_{MAP} &= \arg\max_\theta \left(\log L(\theta|\mu=0,b) + \sum_i \log p(y_i|x_i,\theta)\right) \\ &= \arg\max_\theta \left(\log \frac{1}{2b} \exp\left(-\frac{1}{b}||\theta||_1^2\right) + \sum_i \log p(y_i|x_i,\theta)\right) \\ &= \arg\max_\theta \left(-\frac{1}{b}||\theta||_1^2 + \sum_i \log p(y_i|x_i,\theta)\right) \end{aligned}$$

4. Support Vector Machines

In logistic regression, we maximize the likelihood of assigning the correct target class probability to a group of predictors. However, if our ultimate goal is to choose the most likely class for the group of predictors, we care less about getting the probability perfect when the choice of class is

already determined (i.e. the probability is already fairly high or low), and more about choosing the correct class when the probability is borderline. In this case, we should focus on finding the best separation between the classes.

A Support Vector Machine (SVM) computes the "best" separation between classes as the maximum-margin hyperplane, i.e. the hyperplane which maximizes the distance of the closest points to the border (which are called the support vectors). For hyperplane of the form $w^T x + b = 0$, we can assume $||w|| = 1$ because dividing the equation by $||w||$ yields the same plane, and thus the parameters $\theta = (w, b)$ which yield the maximum margin are given by

$$\begin{aligned}\theta_{MM} &= \arg \max_{||w||=1,b} \min_i ||x_i - x|| \\ &= \arg \max_{||w||=1,b} \min_i ||w^T x_i - w^T x|| \\ &= \arg \max_{||w||=1,b} \min_i \left|w^T x_i + b\right|\end{aligned}$$

Through methods in constrained optimization, this "primal" form of the hyperplane can be reparametrized in "dual" form by the parameters $\alpha = (\alpha_i)$, one for each point in the data, which are chosen as

$$\alpha = \arg \max_{\alpha>0, \alpha^T y=0} \left(||\alpha||_1 - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j \right)$$

and for which the hyperplane is stated as

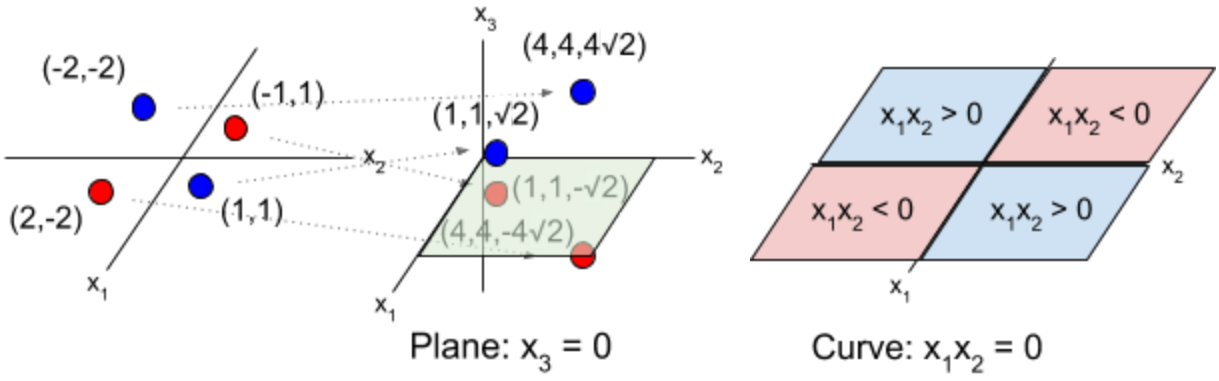
$$\sum_{sv} \alpha_{sv} y_{sv} x_{sv}^T x + b = 0$$

where the sum is taken over the support vectors.

If the data is not linearly separable, then we can use a function $\phi : R^d \to R^{d+n}$ to map the data into a higher dimensional "feature" space before fitting the hyperplane. Below is an example of a function which maps 2-dimensional data $x_i = (x_{i1}, x_{i2})$ into 3-dimensional space.

$$\phi(x_i) = (x_{i1}^2, x_{i2}^2, \sqrt{2} x_{i1} x_{i2})$$

The hyperplane, once projected to the lower-dimensional input space, is able to fit nonlinearities in the data.



Normally, we would worry about blowing up the number of dimensions in the model, which would cause computational and memory problems while training (fitting) the SVM. However, if we choose a function (like the one above) for that can be represented by a "kernel," we can compute the result of dot products in the higher dimensional space, without having to compute and store the values of the data in the higher dimensional space:

$$\begin{aligned}\phi(x_i)^T\phi(x_j) &= (x_{i1}^2, x_{i2}^2, \sqrt{2}x_{i1}x_{i2})^T(x_{j1}^2, x_{j2}^2, \sqrt{2}x_{j1}x_{j2}) \\ &= x_{i1}^2x_{j1}^2 + x_{i2}^2x_{j2}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} \\ &= (x_{i1}x_{j1} + x_{i2}x_{j2})^2 \\ &= (x_i^Tx_j)^2 \\ &= K(x_i, x_j)\end{aligned}$$

This is called the "kernel trick." Some common kernels include the homogeneous polynomial kernel $K(x_i, x_j) = (x_i^Tx_j)^d$, the inhomogeneous polynomial kernel $K(x_i, x_j) = (x_i^Tx_j + 1)^d$, the Gaussian radial basis function (RBF) kernel $K(x_i, x_j) = \exp(ax_i^Tx_j)$, and the hyperbolic tangent kernel $K(x_i, x_j) = \tanh(ax_i^Tx_j + b)$.

Another way to extend the SVM to data which is not linearly separable is to use a soft margin, where we minimize a loss function which penalizes data on the wrong side of the hyperplane. We introduce a "hinge loss" function which is zero for data on the correct side of the margin and is proportional to distance for data on the wrong side of the hyperplane, and minimize the total hinge loss:

$$\theta_{SM} = \arg\min_{||w||=1,b} \sum_i \max\left(0, 1 - y_i(w^Tx_i + b)\right)$$

The hinge loss function is named as such because its graph looks like a door hinge. When the data is linearly separable, the total hinge loss is minimized by our previous "hard margin"

method. In practice, it is difficult to deal with the $||w|| = 1$ constraint, but we want to prevent the weights from blowing up, so we replace the constraint with a regularization:

$$\theta_{SM} = \arg\min_{w,b} \left(-\lambda ||w||^2 + \sum_i \max\left(0, 1 - y_i(w^T x_i + b)\right) \right)$$

A similar dual form exists for this minimization problem, on which the kernel trick is also applicable.

SVMs can be extended to multiclass problems by building a model for each pair of classes and then selecting the class which receives the most votes from the ensemble of models. This is called one-vs-one (OvO) classification.

Another option is to force the SVM to give a probabilistic score rather than binary output, build a model for each class against the rest of the data, and then select the class whose model gives the highest score. This is called one-vs-rest (OvR) classification. To induce a probabilistic score, one can interpret the distance from the hyperplane as the logit (log of the odds ratio) of the in-class probability:

$$\log\left(\frac{p}{1-p}\right) = |w^T x + b|$$

$$p = \frac{1}{1 + \exp(-|w^T x + b|)}$$

SVMs can also be used for regression, in which case they are called Support Vector Regressors (SVRs). In SVRs, the support vectors are the furthest points form the hyperplane, and the task is to minimize the distance to the support vectors.

5. Neural Networks

Neural Networks (NNs) consist of layers of “neurons,” where each neuron has an “activity” which is computed as a function of the weighted sum of activities of neurons in the previous later. The first layer of neurons are activated directly from the data, and the activation of a particular neuron in the last layer represents the likelihood of the data belonging to a particular class. NNs are similar to SVMs in that they project the data to a higher-dimensional space and fit a hyperplane to the data in the projected space. However, whereas SVMs use a predetermined kernel to project the data, NNs automatically construct their own projection by iteratively adjusting (“training”) the weights in the intermediate (“hidden”) layers to minimize a

loss function. Unlike with the kernel trick in SVMs, the training of additional layers in NNs incurs significant additional computational cost, and much work has been devoted to optimizing algorithms and hardware usage to speed up the training of Deep Neural Networks (DNNs) consisting of many hidden layers.

Each layer ℓ of a NN consists of a parameter matrix W_ℓ, where the ith row vector contains the weights received by the ith neuron in the next layer. If we define f_ℓ as the activation function which is applied component-wise (i.e. neuron-wise) at the ℓth layer, and include a bias term as a neuron in each layer whose activity is always 1, then the output activities of the network ℓ layers after the first layer is given by

$$o_\ell(x_i) = f_\ell(W_{n-1}(...W_2 f_1(W_1 x_i)...))$$

We can write this recursively as

$$\begin{aligned} o_0(x_i) &= x_i \\ o_\ell(x_i) &= f_\ell(W_n o_{\ell-1}(x_i)), \ell \geq 1 \end{aligned}$$

When counting layers, we do not count the first layer because it reads in the data and is therefore not associated with trainable weight parameters. This way, each layer is associated to a weight matrix.

For a regression network with n layers, a loss function $L(o_n)$ is chosen to compare the output $o_n(x_i)$ to the desired target y_i from the data. Common choices for this loss function include L1 and L2 error. For a classification network with n layers, we normalize the output to $\widehat{o_n}(x_i) = o_n(x_i)/||o_n(x_i)||$ so that we can interpret it as a probability. Then, a loss function $L(\widehat{o_n})$ is chosen to compare the discrepancy between the output $\widehat{o_n}(x_i)$ and the desired target ("ground truth") y_i from the data. For classification, the loss function is usually chosen as the cross-entropy. For each data point, the cross entropy is given by

$$L(\widehat{o_n}) = -\sum_j \left[y_{ij} \log(\widehat{o_{nj}}) + (1 - y_{ij}) \log(1 - \widehat{o_{nj}}) \right]$$

To make sense of the cross-entropy, notice that it can be simplified to

$$L(\widehat{o_n}) = -\log \left[\widehat{o_{nj}}^{y_{ij}} (1 - \widehat{o_{nj}})^{1-y_{ij}} \right]$$

and if the ground truth is a single class c, i.e. $y_{ic} = 1$ and $y_{ij \neq c} = 0$, then it becomes

$$\begin{aligned} L(\widehat{o_n}) &= -\log\left[\widehat{o_{nc}}\prod_{j\neq c}(1-\widehat{o_{nj}})\right] \\ &= -\log\left[P(\text{in correct class})P(\text{not in wrong class})\right] \end{aligned}$$

The training algorithm, called gradient descent, consists of iteratively updating the weights at each layer according to

$$W_\ell \to W_\ell - \alpha\frac{\partial L}{\partial W_\ell}$$

where α is the learning parameter which governs how quickly the weights change. There are other variations of gradient descent, such as stochastic gradient descent (SGD), where the learning parameter is randomized to assist the weights in breaking out of a shallow minima while allowing them to settle into a deeper minima, and SGD with momentum, which is meant to mimic the trajectory of a ball rolling down a bumpy hill into a valley. The main problem in all of these methods, though, is computing the derivative ("gradient") of the loss function with respect to the weights. Luckily, there is a pattern to it, which we will see after computing the gradient for $\ell = n, n-1, n-2$.

Computing for $\ell = n$, we have

$$\begin{aligned} \frac{\partial L}{\partial W_n} &= L'(o_n)\frac{\partial o_n}{\partial W_n} \\ &= L'(o_n)\cdot f_n'(W_n o_{n-1})\frac{\partial}{\partial W_n}\left[W_n o_{n-1}\right] \\ &= \left[L'(o_n)\cdot f_n'(W_n o_{n-1})\frac{\partial}{\partial W_n}\right] o_{n-1}^T \end{aligned}$$

Where the $(\cdot)$ operation represents the Hadamard product. We define

$$\delta_n = L'(o_n)\cdot f_n'(W_n o_{n-1})$$

so that

$$\frac{\partial L}{\partial W_n} = \delta_n o_{n-1}^T$$

Computing for $\ell = n-1$, we have

$$\begin{aligned}\frac{\partial L}{\partial W_{n-1}} &= \delta_n \frac{\partial}{\partial W_{n-1}} \left[W_n o_{n-1}\right] \\ &= W_n^T \delta_n \frac{\partial o_{n-1}}{\partial W_{n-1}} \\ &= \left[W_n^T \delta_n \cdot f'_{n-1}(W_{n-1} o_{n-2})\right] o_{n-2}^T\end{aligned}$$

We define

$$\delta_{n-1} = W_n^T \delta_n \cdot f'_{n-1}(W_{n-1} o_{n-2})$$

so that

$$\frac{\partial L}{\partial W_{n-1}} = \delta_{n-1} o_{n-2}^T$$

Computing for $\ell = n-2$, we have

$$\begin{aligned}\frac{\partial L}{\partial W_{n-2}} &= \delta_{n-1} \frac{\partial}{\partial W_{n-2}} \left[W_{n-1} o_{n-2}\right] \\ &= \left[W_{n-1}^T \delta_{n-1} \cdot f'_{n-2}(W_{n-2} o_{n-3})\right] o_{n-3}^T\end{aligned}$$

We define

$$\delta_{n-2} = W_{n-1}^T \delta_{n-1} \cdot f'_{n-2}(W_{n-2} o_{n-3})$$

so that

$$\frac{\partial L}{\partial W_{n-2}} = \delta_{n-2} o_{n-3}^T$$

Putting it all together, we have that

$$\frac{\partial L}{\partial W_\ell} = \delta_\ell o_\ell^T$$

where

$$\begin{aligned}\delta_n &= L'(o_n) \cdot f'_n(W_n o_{n-1}) \\ \delta_{\ell-1} &= W_\ell^T \delta_\ell \cdot f'_\ell(W_{\ell-1} o_{\ell-2}), 2 \leq \ell \leq n\end{aligned}$$

This method for computing the gradients is called “backpropagation,” because we propagate the δ_ℓ terms backwards through the layers, from the last layer to the first layer.

The equations for backpropagation also give us insight into our choice of activation function. If we choose a sigmoidal activation function which levels off, then the gradient will vanish for neurons whose activations are too large in magnitude. But if we choose a linear activation function which maintains a slope of 1 everywhere, then we have nothing more than a linear model, and the network is unable to project the data into a higher-dimensional space before fitting the hyperplane. The solution is to use a “rectified” linear unit (ReLU) which is linear for positive inputs, and zero for negative inputs:

$$f(x) = \max(0, x)$$

Ideally, we’d use a “softmax” function which is differentiable at zero unlike ReLUs, but ReLUs are so much faster to compute that we use them anyway. We can usually get away with the slope being zero for negative inputs to the ReLU because the weighted sums in the network tend to be positive sometimes. However, if we set the learning rate too high, we can sometimes end up with neurons whose weighted sums are always negative, and consequently whose gradients and activity are always zero. To overcome this problem of “dead” neurons, one can use leaky ReLUs which have a small gradient and activity even for negative inputs:

$$f(x) = \max(\epsilon x, x)$$

That being said, ReLUs may not be the best choice for the output layer of the network, which is supposed to represent a regression or classification prediction. For regressions, linear activation functions are a better choice in the final layer, and for classifications, softmax units are a better choice in the final layer.

Due to the large number of parameters in NNs, they are prone to overfitting. However, the risk of overfitting can be reduced by “dropout,” a method used to avoid training all of the weights on all of the training data. Dropout involves randomly turning of or “dropping out” neurons from the network during each training iteration, and then keeping the weights of those neurons unchanged during the weight update. Dropout also increases training speed, since dropping out half the neurons in a network cuts the number of computations in half.

One type of neural network that has seen widespread success in the realm of image processing is the convolutional neural network (CNN), which is reduces the number of parameters (thus enabling deep networks of many layers) by taking advantage of spatially local input patterns. In CNNs, each layer of neurons is really a stack of sub-layers, and each neuron in a sub-layer is connected to only a small region (“receptive field”) of a single sub-layer in the preceding layer. Receptive field weights are shared across neurons within a sub-layer, thus forming a template (“convolution”) that can be interpreted as the pattern of activation that the sub-layer is trained to

detect within the sub-layer in the preceding layer. A sub-layer's convolution can be expressed as a weighted sum of different offsets of the convolution in the sub-layer in the preceding layer, and by carrying the weighted sum through all the layers down to the input layer, one can see the visual feature that the sub-layer is trained to detect within the image. Visual features of sub-layers within lower layers are usually simple, like lines and edges, whereas visual features of sub-layers within higher layers can be complex, like faces or cars.

6. Decision Trees

One drawback of SVMs and NNs is that they are black-box models, meaning they are uninterpretable. Although they can model highly nonlinear data, we can't make much sense of what the model has learned by looking at the parameters. On the other hand, the parameters in linear regressions make intuitive sense as the contributions of individual factors to the overall decision, but they are restricted by linearity and thus won't make a good predictive model for highly nonlinear data. Decision trees bridge the gap and are able to model nonlinear data while remaining interpretable.

A decision tree constructs a model by recursively partitioning ("splitting") class data along some value of a predictor ("attribute") until each partition represents a single class of data. The tree starts with a single node which represents all of the data, and then splits into two child nodes to separate the data into two groups which are as homogeneous ("pure") as possible. Then, each child node performs the same splitting process to produce two more child nodes of maximum purity, and so on, until each terminal node ("leaf") of the tree is 100% pure or the data cannot be split any more (sometimes otherwise identical records may have different classes). The predicted probability distribution for the class of any input x is computed as the frequency distribution of classes within the input's corresponding leaf.

The metric which is used to quantify the purity of a split is called the splitting criterion, and it is often chosen as information gain or Gini impurity. Information gain measures the reduction in impurity ("information entropy") achieved by a split. Information entropy for a node is measured by the expectation value

$$\begin{aligned} H &= \left\langle \log\left(\frac{1}{P(y=c)}\right)\right\rangle_c \\ &= -\sum_c P(y=c)\log P(y=c) \end{aligned}$$

over data points (x, y) and classes $c \in \{y\}$ within the node, where $P(y=c)$ is the proportion of data points in the parent node that have $y = c$. Information entropy is largest for uniform distributions, and zero for distributions which are concentrated at a single point. Information gain

is the entropy of the parent node, minus the weighted average entropy of the child nodes (weighted by its proportion of data points from the parent node). Gini impurity is very similar to information entropy, just a little faster to compute. It is given by

$$G = 1 - \sum_c P(y = c)^2$$

To prevent the tree from overfitting the data, which it will almost certainly do if left to construct an unlimited number of partitions, the tree is "pruned." Pruning can be achieved by stopping the tree prior to full growth, in which it is called pre-pruning, or by cutting the tree short after full growth, in which it is called post-pruning. Pre-pruning can be achieved by avoiding splitting a node if the split purity is below some threshold value -- though, any choice of such value is rather ad-hoc. With post-pruning, it is possible to take a more principled approach, using cross-validation to check the effect of pruning on the tree's test accuracy.

7. Ensemble Methods

One big disadvantage of decision trees is that they have a high variance (i.e. they are unstable, not robust to noise in the data). A slight change in the data can cause a different split to occur, giving rise to different child nodes and splits all the way down the tree, potentially leading to different predictions. However, we can make the predictions more stable by averaging them across an ensemble of many different decision trees, called a random forest. Random forests grow a variety of decision trees by forcing each split attribute to be selected from a random subset of candidates. They also train each tree using a random subset of the available training data, which is known as bootstrap aggregating or "bagging" for short.

In general, bagging constructs an ensemble of models which reduces model variance, making it suitable for complex models (low bias, high variance). For simple models (high bias, low variance), another ensemble model called gradient boosting can be used to reduce model bias. Gradient boosting performs gradient descent on a cost function by building the ensemble as a sequence of "error-correcting" models, where each model is trained on a subset of the training data that emphasizes instances that were misclassified by the preceding model. The output of the ensemble is a weighted average, where the weight given to a model's prediction depends on the model's accuracy.

Another example of an ensemble that we encountered earlier was the Bayes optimal ensemble, which averages over all sets of models with a given set of parameters. Since the average is difficult to compute, we settled for choosing the single model which contributed most to the ensemble (MAP/MLE). However, are ways to approximate the average through sampling, known as Bayesian Model Combination (BMC).

The type of ensemble model that wins most data science competitions, perhaps surprisingly, is not the Bayes optimal ensemble. Rather, it is the stacked model, which consists of an ensemble of entirely different species of models together with some combiner algorithm (usually chosen as a logistic regression) which is trained to make a final prediction using the predictions of the models within the ensemble as additional inputs. Although the Bayes optimal ensemble performs at least as well as (and often better than) stacking when the correct data-generating model is on the list of models under consideration, the correct data-generating model for data difficult enough to warrant a competition is often too complex to be approximated by models in the Bayes optimal ensemble. In such cases, it is advantageous to have a diverse basis of models from which to approximate the data-generating model.